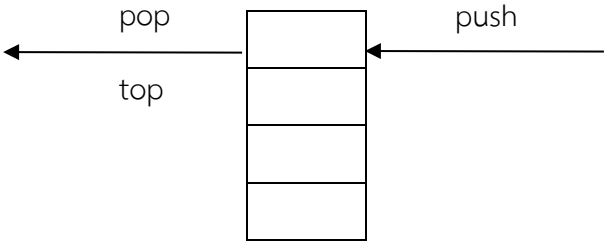
3. สแตก (Stack)

สแตกเป็นโครงสร้างข้อมูลที่มีการเพิ่มและดึงข้อมูลที่ปลายข้างเดียวของโครงสร้างข้อมูล คุณสมบัตินี้เรียกว่า last in first out (LIFO) คือ ข้อมูลที่เข้าไปในสแตกหลังสุดจะถูกนำออกจากสแตกเป็นอันดับแรก โดยการนำข้อมูลเข้าไปในสแตกเรียกว่า **push** และการดึงข้อมูลออกจากสแตกจะเรียกว่า **pop**

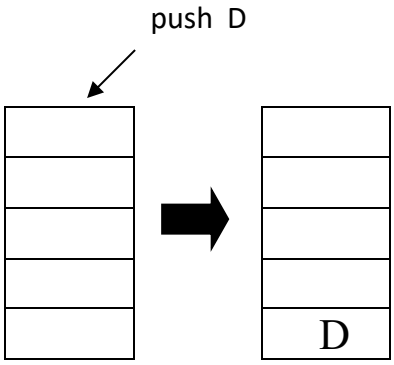
โครงสร้างข้อมูล stack ใน STL

ในโครงสร้างข้อมูลแบบ stack ใน STL มีตัวชี้ที่ตำแหน่งบนสุดของ stack ที่เป็นตำแหน่งในการเพิ่มข้อมูลเข้า หรือ push การดึงข้อมูลออก หรือ pop และ top หมายถึงการเข้าถึงข้อมูลโดยไม่ดึงข้อมูลออกจาก stack

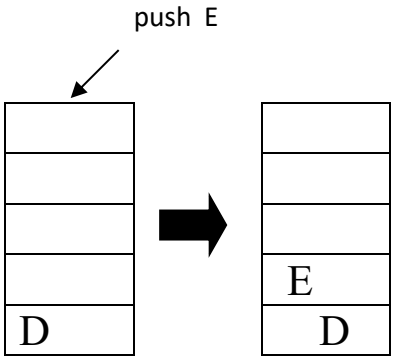


รูปที่ 3.1 ลักษณะโครงสร้างข้อมูลแบบ stack

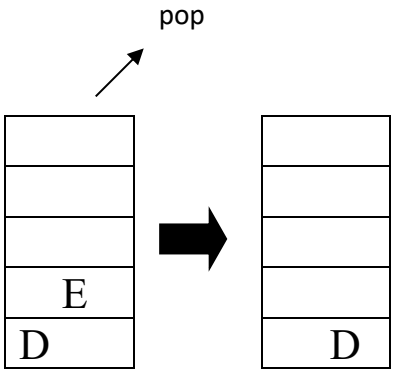
ตัวอย่างปฏิบัติการของ stack ในการเพิ่มข้อมูล และการดึงข้อมูล



รูปที่ 3.2 ก่อนและหลังการ push D เข้าสแตก



รูปที่ 3.3 ก่อนและหลังการ push E เข้าสแตก



รูปที่ 4 ก่อนและหลังการ pop E ออกจากสแตก

การประกาศการใช้ stack และฟังก์ชันที่ใช้

Operation	Effect
`stack <type> c`	Creates an empty list without any elements
`c.size()`	Returns the actual number of elements
`c.push()`	Inserts an element into the stack
`c.top()`	Returns the top element in the stack without removing from the stack
`c.pop()`	Removes an element from the stack
`c.empty()`	Determines whether a stack is empty

ตัวอย่างที่ 3.1 การใช้ operation พื้นฐานของ stack

```
1.   #include <iostream>
2.   #include <stack>
3.
4.   using namespace std;
5.
6.   int main() {
7.
8.      int i,j;
9.      stack <int>st;
10.     cin >> i;
11.     st.push(i);
12.
13.     cin >> j;
14.     st.push(j);
15.
16.     cout << "***Output***\n";
17.     cout << st.top() ;
18.     st.pop();
19.     cout << endl;
20.     cout << st.top() ;
21.     return 0;
22.  }
```

ข้อมูลนำเข้า

```
1
2
```

ผลลัพธ์

```
***Output***
2
1
```

ตัวอย่างที่ 3.2 การวนทำซ้ำเพื่อแสดงค่าใน stack

```
1.   #include <iostream>
2.   #include <stack>
3.   using namespace std;
4.
5.   int main() {
6.
7.      int n;
8.      string word;
9.      stack <string>st;
10.
11.     cin>>n;
12.     for (int i=0;i<n;i++)
13.     {
14.        cin>>word;
15.        st.push(word);
16.     }
17.
18.     while (!st.empty())
19.     {
20.        cout<<st.top()<<endl ;
21.        st.pop();
22.     }
23.     return 0;
24.  }
```

ข้อมูลนำเข้า

```
4
Word1
Word2
Word3
Word4
```

ผลลัพธ์

```
***Output***
Word4
Word3
Word2
Word1
```

การแปลงนิพจน์จาก infix เป็น postfix

ตัวอย่างนิพจน์ `A + B * C – D`

Symbol	Stack	Output	Action
A	empty	A	'A' printed
+	+	A	'+' pushed
B	+	A B	'B' printed
*	* +	A B	'*' pushed
C	* +	A B C	'C' printed
–	–	A B C * +	'*' popped & printed '+' popped & printed '–' pushed
D	–	A B C * + D	'D' printed
empty		A B C * + D –	All operators in stack popped

การแปลงนิพจน์จาก infix เป็น postfix (กรณีมีวงเล็บ)

ตัวอย่างนิพจน์ `A + B * (C – D) / E`

Scaned symbol	Stack	Output
A	empty	A
+	+	A
B	+	A B
*	* +	A B
(	(* +	A B
C	(* +	A B C
–	– (* +	A B C
D	– (* +	A B C D
)	* +	A B C D –
/	/ +	A B C D – *
E	/ +	A B C D – * E
	empty	A B C D – * E / +

การแปลงนิพจน์จาก postfix เป็น infix

ตัวอย่างนิพจน์ `A B C – + D E – F G – H + / *`

Symbol	Stack
A	A
B	B A
C	C B A
–	B – C A
+	A + B – C
D	D A + B – C
E	E D A + B – C
–	D – E A + B – C
F	F D – E A + B – C
G	G F D – E A + B – C
–	F – G D – E A + B – C

H	H F - G D - E A + B - C
+	F — G + H D - E A + B - C
/	(D - E) / (F — G + H) A + B - C
*	(A + B — C) * (D — E) / (F — G + H)
empty	(A + B — C) * (D — E) / (F — G + H)

ที่มา: http://scanftree.com/Data_Structure/postfix-to-infix

การแปลงนิพจน์จาก postfix เป็น infix

ตัวอย่างนิพจน์ `A B D + – E F + *`

Symbol	Stack

Computer Olympic 2022

Notation of Expression (using stack)

Notation of Expression

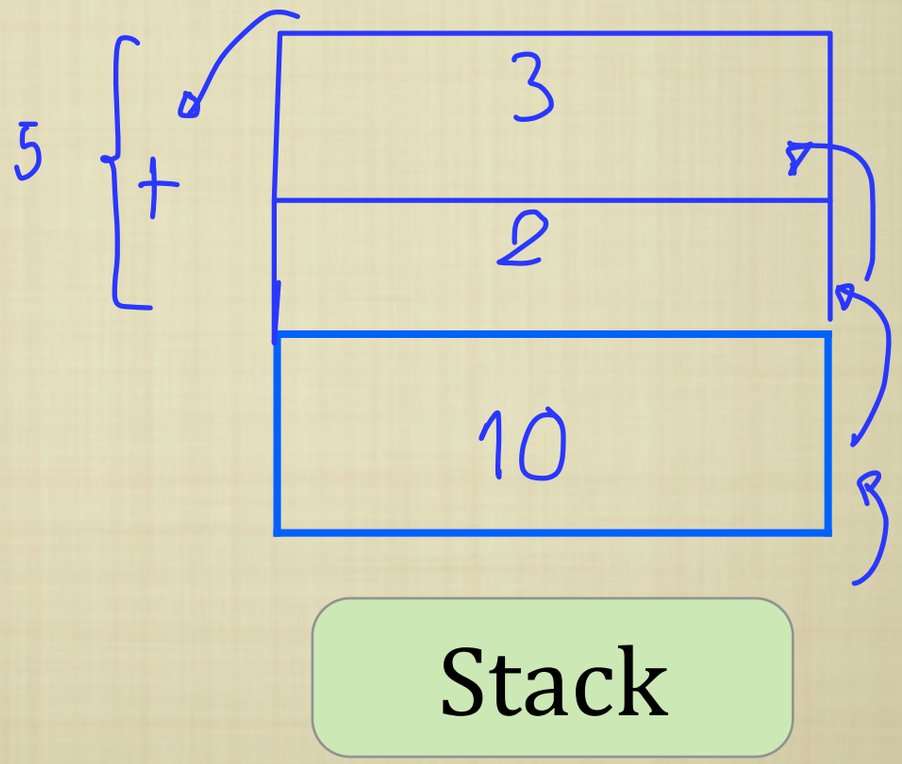
`X + Y`	Infix notation
`+ X Y`	Prefix notation
`X Y +`	Postfix notation

Stack operation: postfix

```
10 2 3 + /
```

คือ 3, 2

แล้วมาบวก

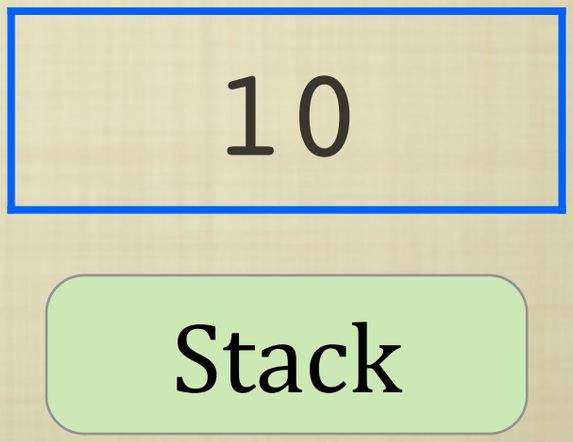


Stack

Stack operation: postfix

```
10 2 3 + /
```

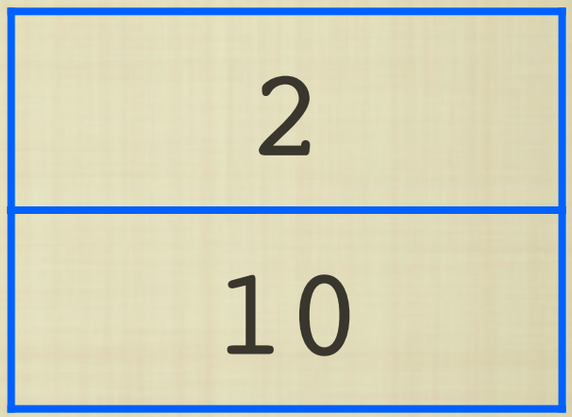
```
PUSH 10
```



Stack operation: postfix

```
10 2 3 + /
```

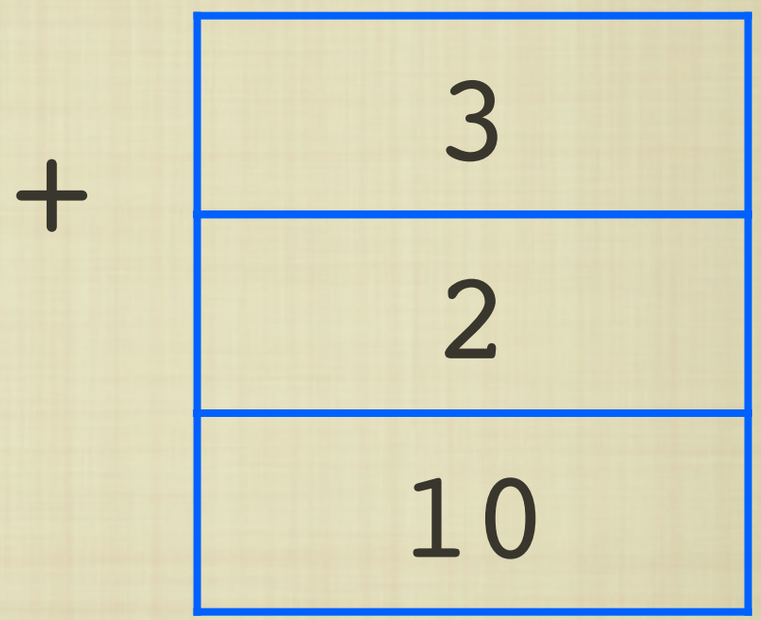
```
PUSH 2
```



Stack operation: postfix

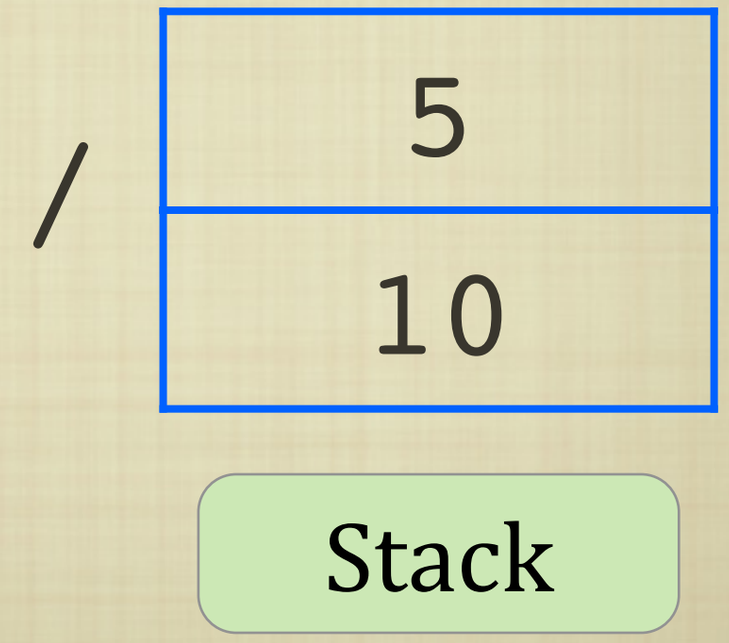
```
10 2 3 + /
```

```
PUSH 3
+
```



Stack operation: postfix

```
10 2 3 + /
```



Stack operation: postfix

```
10 2 3 + /
```

Stack

Stack operation: infix

```
( 2 * ( ( 2 + 3 ) + 4) )
```

Dijkstra's two-stack evaluation

- push operands onto the operand stack
- push operators onto the operator stack
- ignore left parentheses
- on encountering a right parentheses
  - pop an operator
  - pop operands (top 2 values)
  - push result onto the operand stack

Stack operation: infix

```
( 2 * ( ( 2 + 3 ) + 4 ) )
```

symbol	operand stack	operator stack	action
(	empty	empty	ignore '('
2	2	empty	push 2
*	2	*	push *
(	2	*	ignore '('
(	2	*	ignore '('
2	2 2	*	push 2
+	2 2	* +	push +
3	2 2 3	* +	push 3

Stack operation: infix

```
( 2 * ( ( 2 + 3 ) + 4 ) )
```

symbol	operand stack	operator stack	action
3	2 2 3	* +	push 3
)	2 5	*	pop operator pop operands push result
+	2 5	* +	push '+'
4	2 5 4	* +	push 4
)	2 9	*	pop operator pop operands push result
)	**18**		

Postfix-to-Infix

1. scan expression from left to right
2. if the scanned character is an operand, **push it to the stack**
3. if the scanned character is an operator:
    3.1 pop the top 2 values from stack
        - **check the precedence of the scanned operator and the operator in the stack**
    3.2 put the operator in the middle to create a string
    3.3 push the string back to stack

Postfix-to-Infix

```
struct Token {
   string exp;
   char op;
};

int main()
{
   int prec[200];
   string p;
   stack <Token> s;
   prec['*'] = 3;
   prec['/'] = 3;
   prec['+'] = 2;
   prec['-'] = 2;
   prec['x'] = 5;

   getline(cin,p);
```

Infix to postfix

- if it is an operand, **place** it to the **output**.
- if it is an operator or left parentheses “(”, **place** it onto the **stack**.
- if it is right parentheses “)”, **pop** the stack until we **find left parentheses**.
- if it is an operator, pop entries from the stack until we find an entry of **lower priority** or **left parentheses** “(”.

source from: Mark A. Weiss, “Data structures and algorithm analysis in C++